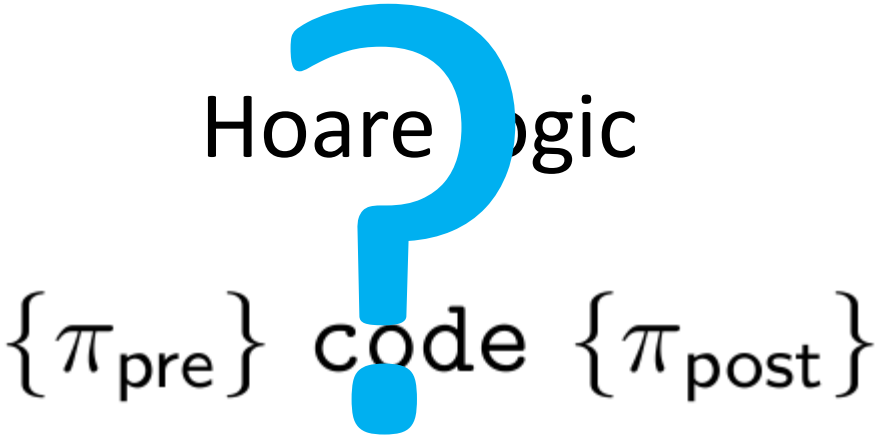
Matching Logic

Grigore Rosu
University of Illinois at Urbana-Champaign

Joint work with Andrei Stefanescu and Chucky Ellison. Started with Wolfram Schulte at Microsoft Research in 2009

Question

… could it be that, after 40 years of program verification, we still lack the right semantically grounded program verification foundation?



Current State-of-the-Art in Program Analysis and Verification

Consider some programming language, L

- Formal semantics of L
  - Typically skipped: considered expensive and useless
- Model checkers for L
  - Based on some adhoc encodings/models of L
- Program verifiers for L
  - Based on some other adhoc encodings/models of L
- Runtime verifiers for L
  - Based on yet another adhoc encodings/models of L
- …

Example of C Program

- What should the following program evaluate to?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

- According to the C “standard”, it is **undefined**
- GCC4, MSVC: it returns **4**
  GCC3, ICC, Clang: it returns **3**
  By April 2011, both Frama-C (with its Jessie verification plugin) and Havoc "prove" it returns **4**

A Formal Semantics Manifesto

- Programming languages must have formal semantics! (period)
  - And analysis/verification tools should build on them
- Informal manuals are not sufficient
  - Manuals typically have a formal syntax of the language (in an appendix)
  - Why not a formal semantics appendix as well?

Motivation and Goal

- We are facing a semantic chaos
  - Operational, denotational, axiomatic, etc.
  - Problematic when dealing with large languages
- Why so many semantic styles?
  - Since none of them is ideal, they have limitations

- We want a powerful, unified foundation for programming language semantics and verification
  - One semantics to serve all the purposes!

Minimal Requirements for an Ideal Language Semantic Framework

- Should be expressive
  - Substitution or environment-based definitions, abrupt control changes (callcc), concurrency, etc.
- Should be executable
  - So we can test it and use it in tools (symb. exec.)
- Should be modular/compositional (thus scale)
  - So each feature is defined once and for all
- Should serve as a program logic
  - So we can also prove programs correct with it

Current Semantic Approaches -Structural Operational Semantics-

- Executable
- Not very modular/compositional (except MSOS)
- Not appropriate for verification
- Only interleaving semantics

Current Semantic Approaches -Evaluation Contexts-

- Executable
- Modular, deals better with control
- Very syntactic, rigid
  - Enforces substitution-based definitions
  - Unsuitable for environment-based definitions
- Not appropriate for verification
- Only interleaving semantics

Current Semantic Approaches -Denotational Semantics-

Reasonable trade-offs

- Mathematical model, somehow compositional
- Not very executable
  - Norish's C semantics is not executable
  - factorial(5) crashes Papaspyrou's C semantics
- Not very good for verification
- Poor for concurrency
- Requires expert knowledge

Current Semantic Approaches -The Chemical Abstract Machine-

CHAM

- Intuitive computational model
- True concurrency (like in nature)
- No machine support
  - It would be very hard, because of its airlock
- Not appropriate for verification

Current Semantic Approaches -Floyd-Hoare Logic-

- Good for program verification
- Requires encodings of structural program configuration properties as predicates
  - Heap, stacks, input/output, etc.
  - Framing is hard to deal with
- Not based on a formal executable semantics
  - Thus, hard to test
  - Semantic errors found by proving wrong properties
  - Soundness rarely or never proved in practice
- Implementations of Floyd-Hoare verifiers for real languages still an art, who few master

Towards a Better Semantic Approach

Starting Point: Rewriting Logic

Meseguer (late 80s, early 90s)

- Expressive
  - Any logic can be represented in RL (it is reflective)
- Executable
  - Quite efficiently; Maude often outperforms SML
- Modular
  - Allows rules to only “match” what they need
- Can potentially serve as a program logic
  - Admits initial model semantics, so it is amenable for inductive or fixed-point proofs

Rewriting Logic Semantics Project

- Project started jointly with Meseguer in 2003-4
- Idea: Define the semantics of a programming language as a rewrite theory (set of rules)
- Showed that most executable semantics approaches can be framed as rewrite logic semantics (Modular/SmallStep/BigStep SOS, evaluation contexts, continuation-based, etc.)
  - But they still had their inherent limitations
- Appropriate techniques/methodologies needed

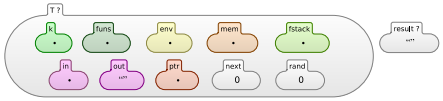
The K Framework
k-framework.org

- A tool-supported rewrite-based framework for defining programming language semantics
- Inspired from rewriting logic
- Used regularly in teaching undergraduate courses
- Main ideas:
  - Represent program configurations as a potentially nested structure of cells (like in the CHAM)
  - Flatten syntax into special computational structures (like in refocusing for evaluation contexts)
  - Define the semantics of each language construct by semantic rules (a small number, typically 1 or 2)

Complete K Definition of KernelC

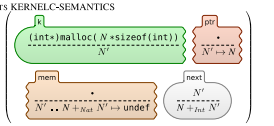
```
MODULE KERNELC-SYNTAX
  IMPORTS K-LATEX+PL-ID+PL-INT
  SYNTAX  Exp ::= Exp + Exp [strict]
                | DeclId
                | Id
                | Int
                | Exp - Exp [strict]
                | Exp ++
                | Exp == Exp [strict]
                | Exp != Exp [strict]
                | Exp <= Exp [strict]
                | Exp < Exp [strict]
                | Exp % Exp [strict]
                | ! Exp
                | Exp && Exp
                | Exp ? Exp : Exp
                | Exp || Exp
                | printf("%d;", Exp ) [strict]
                | scanf("%d",& Exp )
                | scanf("%d", Exp ) [strict]
                | NULL
                | PointerId
                | (int*)malloc( Exp *sizeof(int)) [strict]
                | free( Exp ) [strict]
                | * Exp [strict]
                | Exp [ Exp ]
                | Exp = Exp [strict(2)]
                | Id ( List{Exp} ) [strict(2)]
                | Id ( )
                | random()
                | srandom( Exp ) [strict]
  SYNTAX  Stmt ::= Exp ; [strict]
                | {}
                | { StmtList }
                | if( Exp ) Stmt
                | if( Exp ) Stmt else Stmt [strict(1)]
                | while( Exp ) Stmt
                | return Exp ; [strict]
                | DeclId List{DeclId} { StmtList }
                | #include< StmtList >
  SYNTAX  StmtList ::= StmtList StmtList
                | Stmt
  SYNTAX  Pgm ::= StmtList
  SYNTAX  Id ::= main
  SYNTAX  PointerId ::= * PointerId [ditto]
                | Id
  SYNTAX  DeclId ::= int Exp
                | void PointerId
  SYNTAX  StmtList ::= stdio.h
                | stdlib.h
  SYNTAX  List{Bottom} ::= List{Bottom} , List{Bottom} [assoc hybrid id: () strict]
                | ()
                | Bottom
  SYNTAX  List{PointerId} ::= List{PointerId} , List{PointerId} [ditto]
                | List{Bottom}
                | PointerId
  SYNTAX  List{DeclId} ::= List{DeclId} , List{DeclId} [ditto]
                | DeclId
                | List{Bottom}
  SYNTAX  List{Exp} ::= List{Exp} , List{Exp} [ditto]
                | Exp
                | List{DeclId}
                | List{PointerId}
END MODULE

MODULE KERNELC-DESUGARED-SYNTAX
  IMPORTS K-LATEX
  IMPORTS KERNELC-SYNTAX
  MACRO  ! E = E ? 0 : 1

  MACRO  E1 && E2 = E1 ? E2 : 0

  MACRO  E1 || E2 = E1 ? 1 : E2

  MACRO  if( E ) St = if( E ) St else {}

  MACRO  NULL = 0

  MACRO  I () = I ( () )

  MACRO  int * PointerId = int PointerId

  MACRO  #include< Stmts > = Stmts

  MACRO  E1 [ E2 ] = * E1 + E2

  MACRO  scanf("%d",& * E ) = scanf("%d", E )

  MACRO  int * PointerId = E = int PointerId = E

  MACRO  int X = E ; = int X ; X = E ;

  MACRO  stdio.h = {}

  MACRO  stdlib.h = {}

END MODULE
```

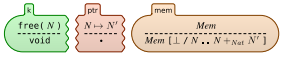
```
MODULE KERNELC-SEMANTICS
  IMPORTS K-SHARED
  IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
  CONFIGURATION:
```




```
  RULE  X  V        env: X ↦ V

  RULE  X ++  I     env: X ↦ I  →  I +Int 1

  RULE  X = V  V    env: X ↦ _  →  V

  RULE  I1 + I2 → I1 +Int I2

  RULE  I1 - I2 → I1 −Int I2

  RULE  I1 % I2 → I1 %Int I2        when I2 !=Int 0

  RULE  I1 <= I2 → Bool2Int ( I1 ≤Int I2 )

  RULE  I1 < I2 → Bool2Int ( I1 <Int I2 )

  RULE  I1 == I2 → Bool2Int ( I1 ==Int I2 )

  RULE  I1 != I2 → Bool2Int ( I1 !=Int I2 )

  RULE  _?_:_ → if(_)_else_

  RULE  if( I ) – else St → St          when I ==Int 0

  RULE  if( I ) St else – → St          when ¬Bool I ==Int 0

  RULE  while( E ) St  /  if( E ) { St while( E ) St } else {}

  RULE  printf("%d;", I )  /  void        out: S  /  S +String Int2String ( I ) +String ";"

  RULE  scanf("%d", N )  /  * N = I        in: I  /  .

  RULE  scanf("%d",& X )  /  X = I         in: I  /  .

  RULE  V ; → •

  RULE  { Sts } → Sts

  RULE  {} → •

  RULE  St Sts → St ↷ Sts

  RULE  int X Xl { Sts }  /  .        funs: .  /  X ↦ int X Xl { Sts }

  RULE  void X Xl { Sts }  /  int X        Sts return void ;
```

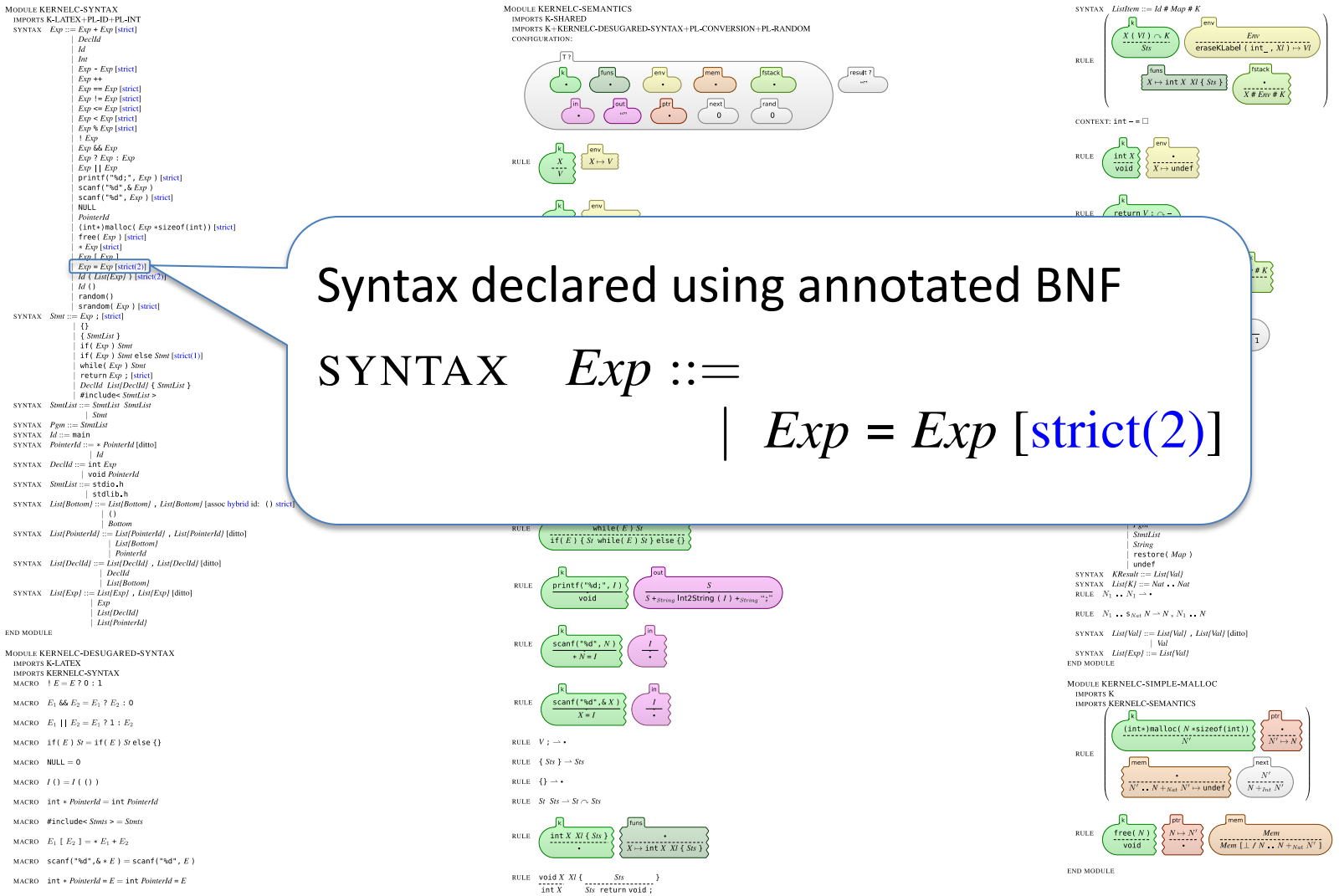
```
  SYNTAX  ListItem ::= Id # Map # K

  RULE  k: X ( Vl ) ↷ K  /  Sts        env: Env  /  eraseKLabel ( int _ , Xl ) ↦ Vl
        funs: X ↦ int X Xl { Sts }        fstack: .  /  X # Env # K

  CONTEXT: int – = □

  RULE  k: int X  /  void        env: .  /  X ↦ undef

  RULE  k: return V ; ↷ –  /  V

  RULE  k: V ↷ .  /  K        env: –  /  Env        fstack: – # Env # K  /  .

  RULE  k: random()  /  randomRandom ( N' )        rand: N'  /  N' +Nat 1

  RULE  k: srandom( I )  /  void

  CONTEXT: * □ = –

  CONTEXT: * □ ++

  SYNTAX  Val ::= Int
                | void
  SYNTAX  Exp ::= Val
  SYNTAX  K ::= List{DeclId}
                | List{Exp}
                | List{PointerId}
                | Pgm
                | StmtList
                | String
                | restore( Map )
                | undef
  SYNTAX  KResult ::= List{Val}
  SYNTAX  List{K} ::= Nat .. Nat
  RULE  N1 .. N1 → •

  RULE  N1 .. sNat N → N , N1 .. N

  SYNTAX  List{Val} ::= List{Val} , List{Val} [ditto]
                | Val
  SYNTAX  List{Exp} ::= List{Val}
END MODULE

MODULE KERNELC-SIMPLE-MALLOC
  IMPORTS K
  IMPORTS KERNELC-SEMANTICS
```





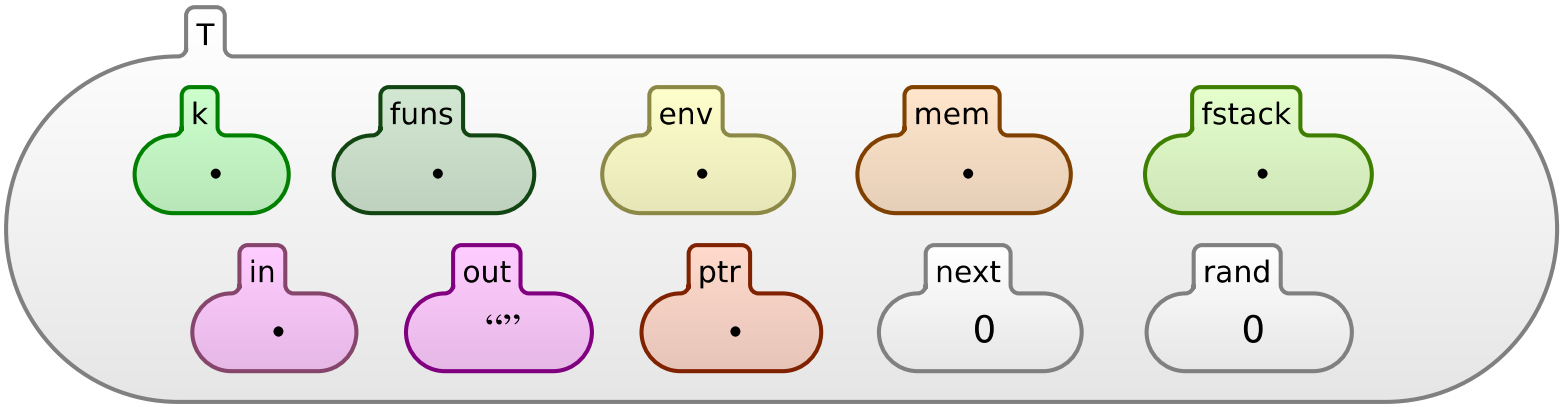

```
END MODULE
```

Complete K Definition of KernelC

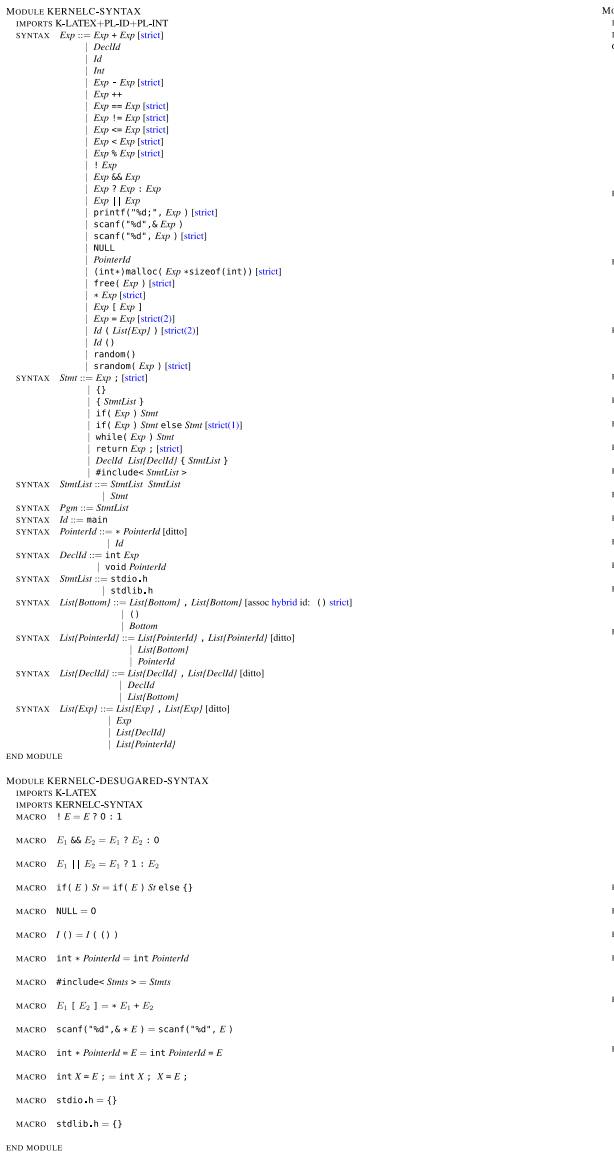


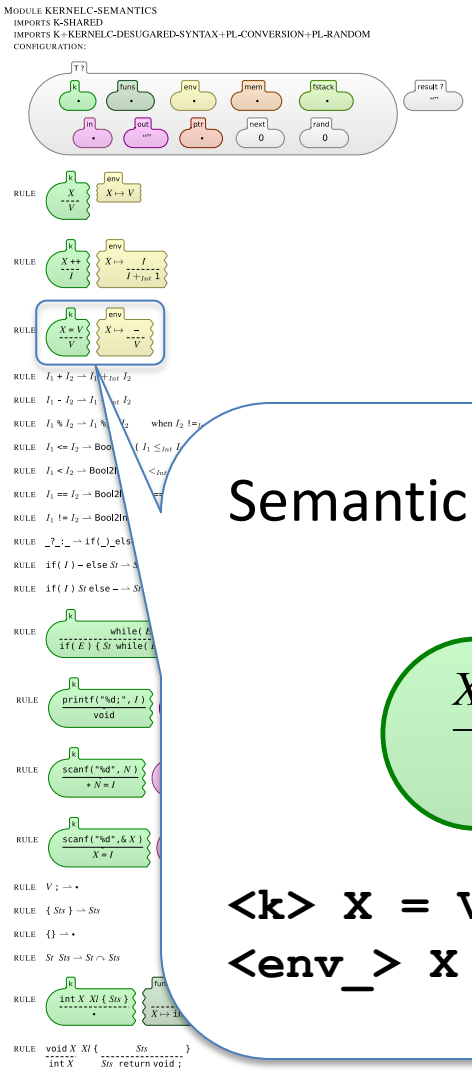
Complete K Definition of KernelC

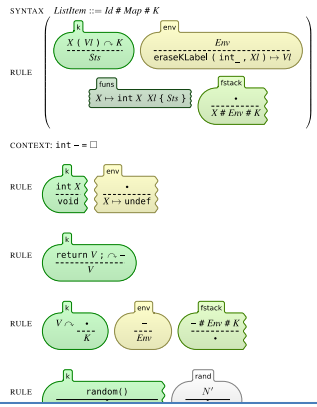
Configuration given as a nested cell structure.
Leaves can be sets, multisets, lists, maps, or syntax



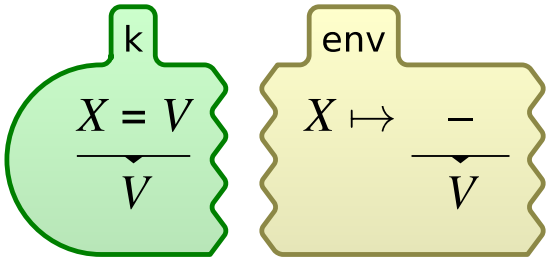
Complete K Definition of KernelC







Semantic rules given contextually




```
<k> X = V => V </k>
<env> X |-> (_ => V) </env>
```

K Semantics are Useful

- Executable, help language designers
- Make teaching PL concepts hands-on and fun
- Currently compiled into
  - Maude, for execution, debugging, model checking
  - Latex, for human inspection and understanding
- Soon to be compiled to
  - OCAML, for fast execution
  - COQ, for meta-property verification

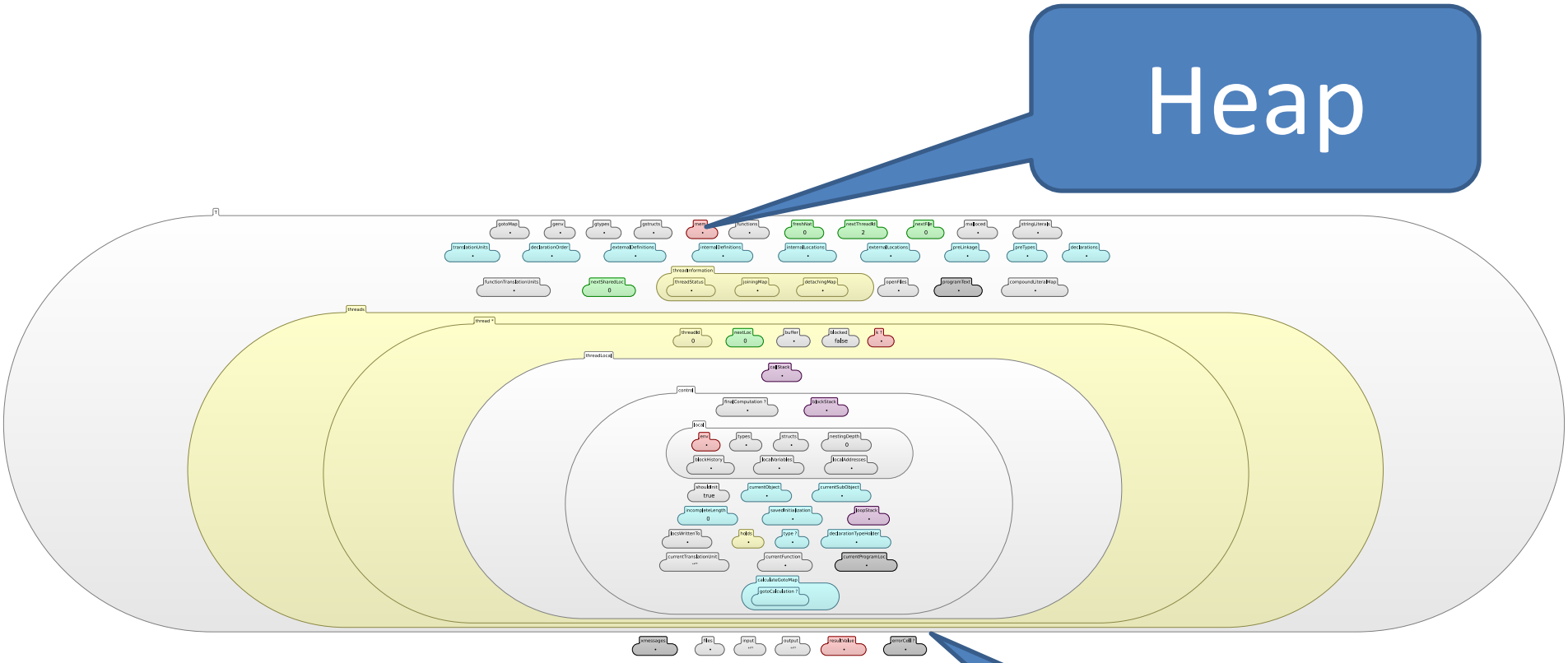
K Scales

Besides smaller and paradigmatic teaching languages, several larger languages were defined

- Scheme : by Pat Meredith
- Java 1.4 : by Feng Chen
- Verilog : by Pat Meredith and Mike Katelman
- C : by Chucky Ellison

etc.

The K Configuration of C



Statistics for the C definition

- Total number of rules: **~1200**

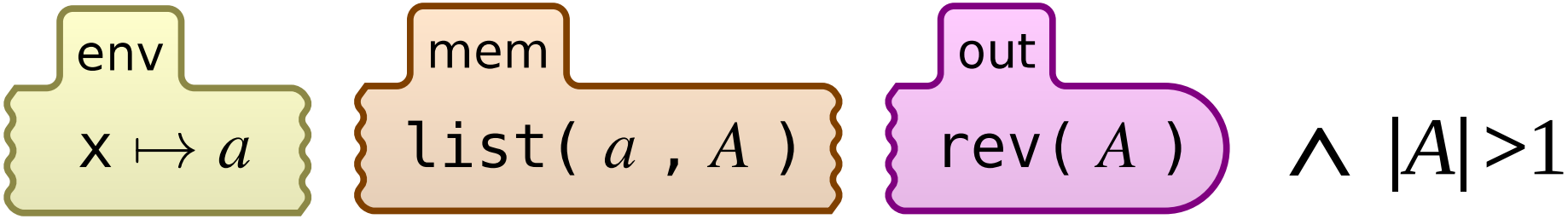
- Has been tested on thousands of C programs (several benchmarks, including the gcc torture test, code from the obfuscated C competition, etc.)
  - Passed **99.2%** so far!
  - GCC 4.1.2 passes 99%, ICC 99.4%, Clang 98.3 (no opt.)
- *The most complete formal C semantics*
- Took more than 18 months to define …
  - Wouldn't it be uneconomical to redefine it in each tool?

Matching Logic = K + FOL

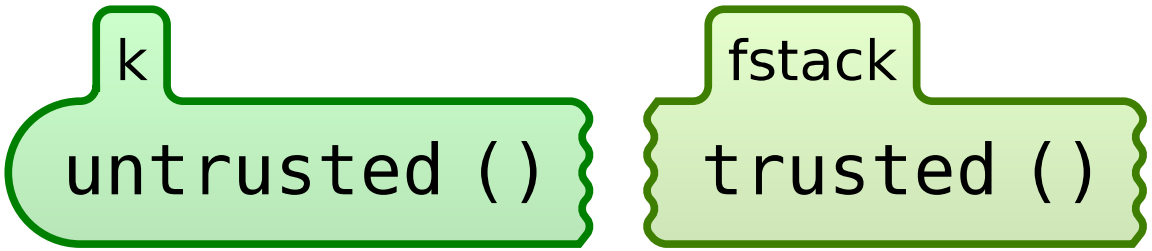
- A logic for reasoning about configurations
- Formulae
  - FOL over configurations, called patterns
  - Configurations are allowed to contain variables
- Models
  - Ground configurations
- Satisfaction
  - Matching for configurations, plus FOL for the rest

Examples of Patterns

- x points to sequence *A* with $|A|>1$, and the reversed sequence rev(*A*) has been output



- **`untrusted()`** can only be called from **`trusted()`**



Matching Logic vs. Separation Logic

- Matching logic achieves separation through matching at the structural (term) level, not through special logical connectives (*)
- Matching logic realizes separation at all levels of the configuration, not only in the heap
  - the heap was only 1 out of the 75 cells in C's def.
- Matching logic can stay within FOL, while separation logic needs to extend FOL
  - Thus, we can use the existing SMT provers, etc.

Matching Logic as a Program Logic

- Hoare style - not recommended

$$\{\pi_{\text{pre}}\}\ \texttt{code}\ \{\pi_{\text{post}}\}$$

  – One has to redefine the PL semantics – impractical

- Rewriting (or K) style – recommended

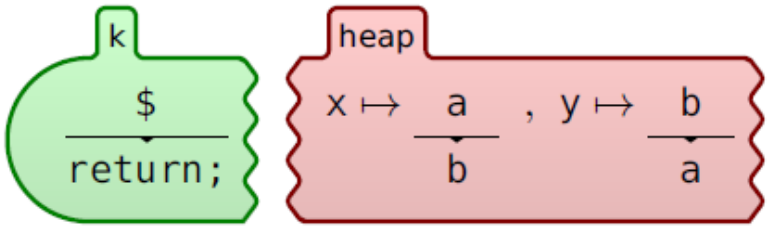
$$left[\texttt{code}] \rightarrow right$$

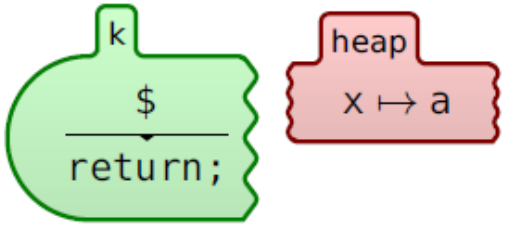
  – One can reuse existing K semantics – very good

Example – Swapping Values

```
void swap(int *x, int *y)
{
  int t;
  t=*x;
  *x=*y;
  *y=t;
}
```

$ = the body





if x = y

- What is the K semantics of the swap function?
- Let $ be its body

```
rule <k> $ => return; <_/k>
     <heap_>
       x|->(a=>b),
       y|->(b=>a)
     <_/heap>
```

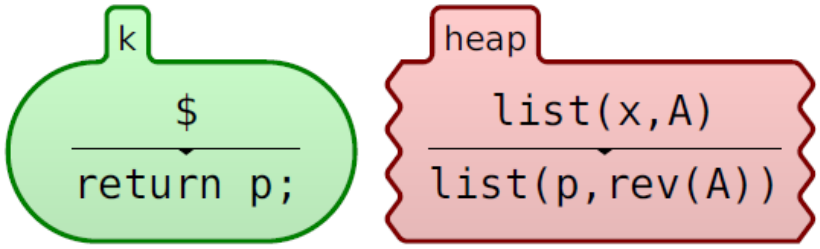
```
rule <k> $ => return; <_/k>
     <heap_> x|-> a <_/heap>
  if x = y
```

Example – Reversing a list

```
struct listNode* reverse(struct listNode *x)
{
  struct listNode *p;
  struct listNode *y;
  p = 0 ;
  while(x) {
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}
```

$

- What is the K semantics of the reverse function?
- Let $ be its body




```
rule <k> $ => return p; </k>
     <heap_> list(x,A) => list(p,rev(A)) <_/heap>
```

Partial Correctness

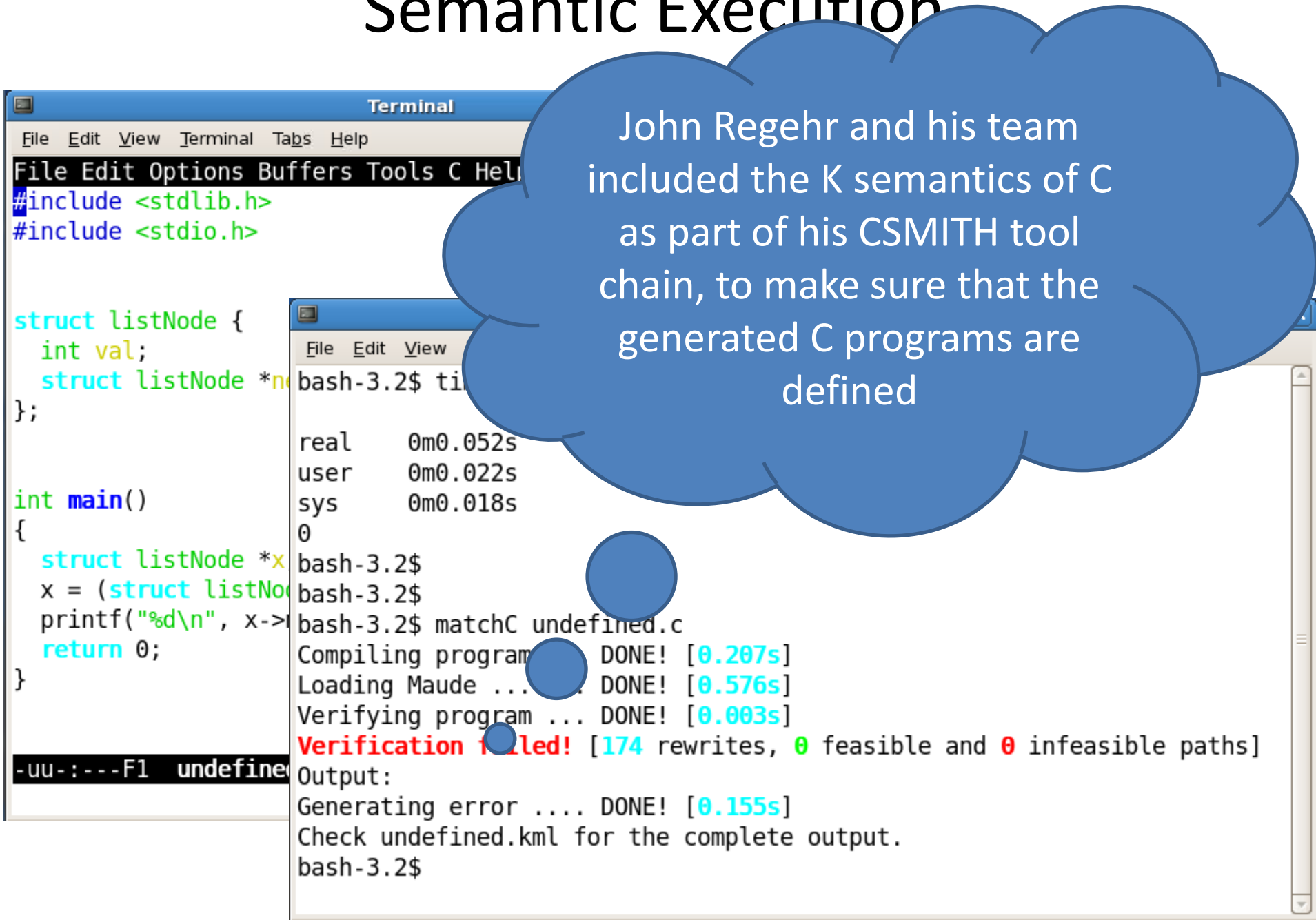
- We have two rewrite relations on configurations
  - $\rightarrow$ given by the language K semantics; safe
  - $\rightarrow$ given by specifications; unsafe, has to be proved
- Idea (simplified for deterministic languages):
  - Pick left $\rightarrow$ right. Show that always left $\rightarrow (\rightarrow \cup \rightarrow)^*$ right modulo matching logic reasoning (between rewrite steps)
- Theorem (soundness):
  - If left $\rightarrow$ right and "config *matches* left" such that config has a normal form for $\rightarrow$, then "nf(config) *matches* right"

MatchC Tool DEMO

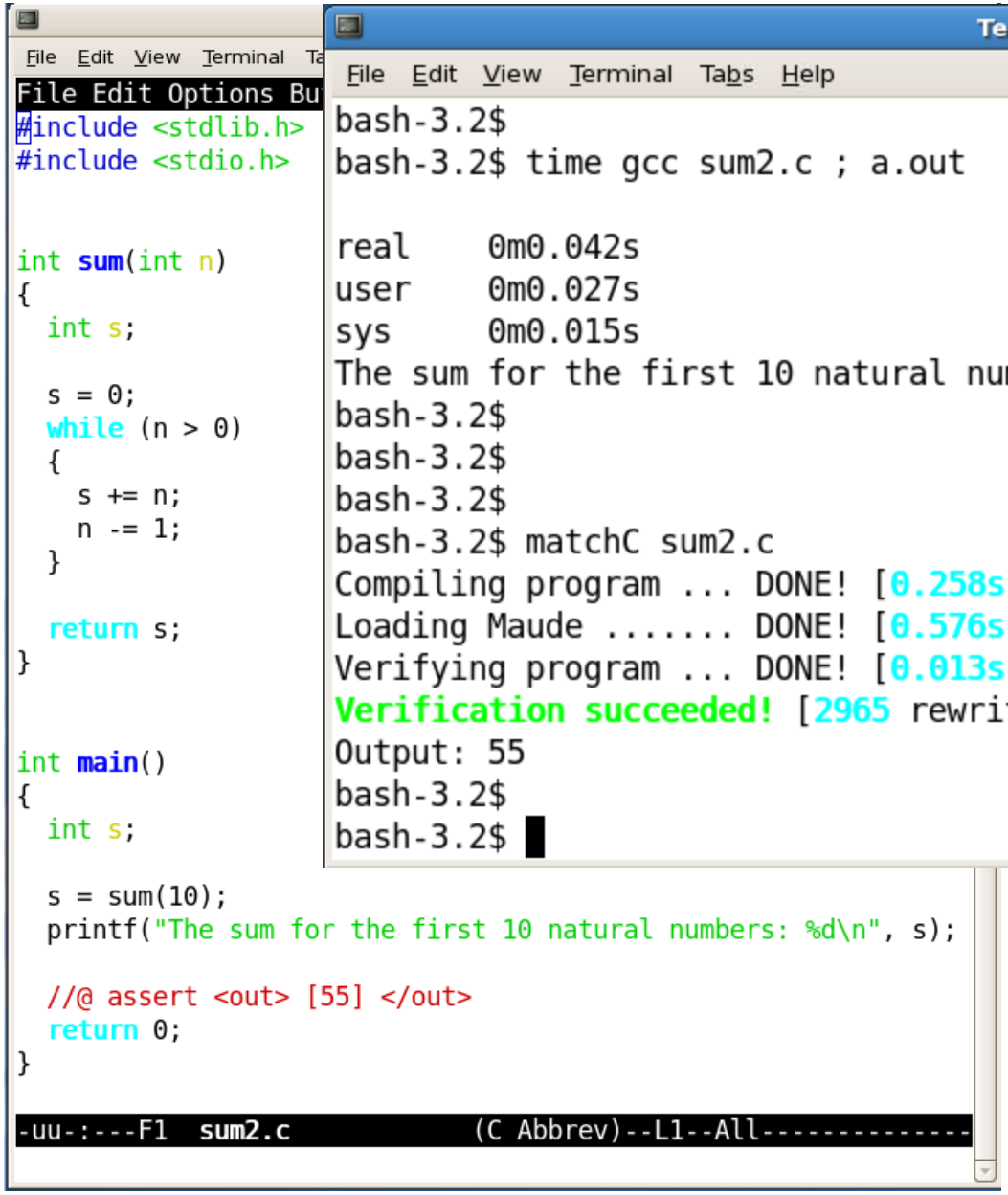
try it online first at

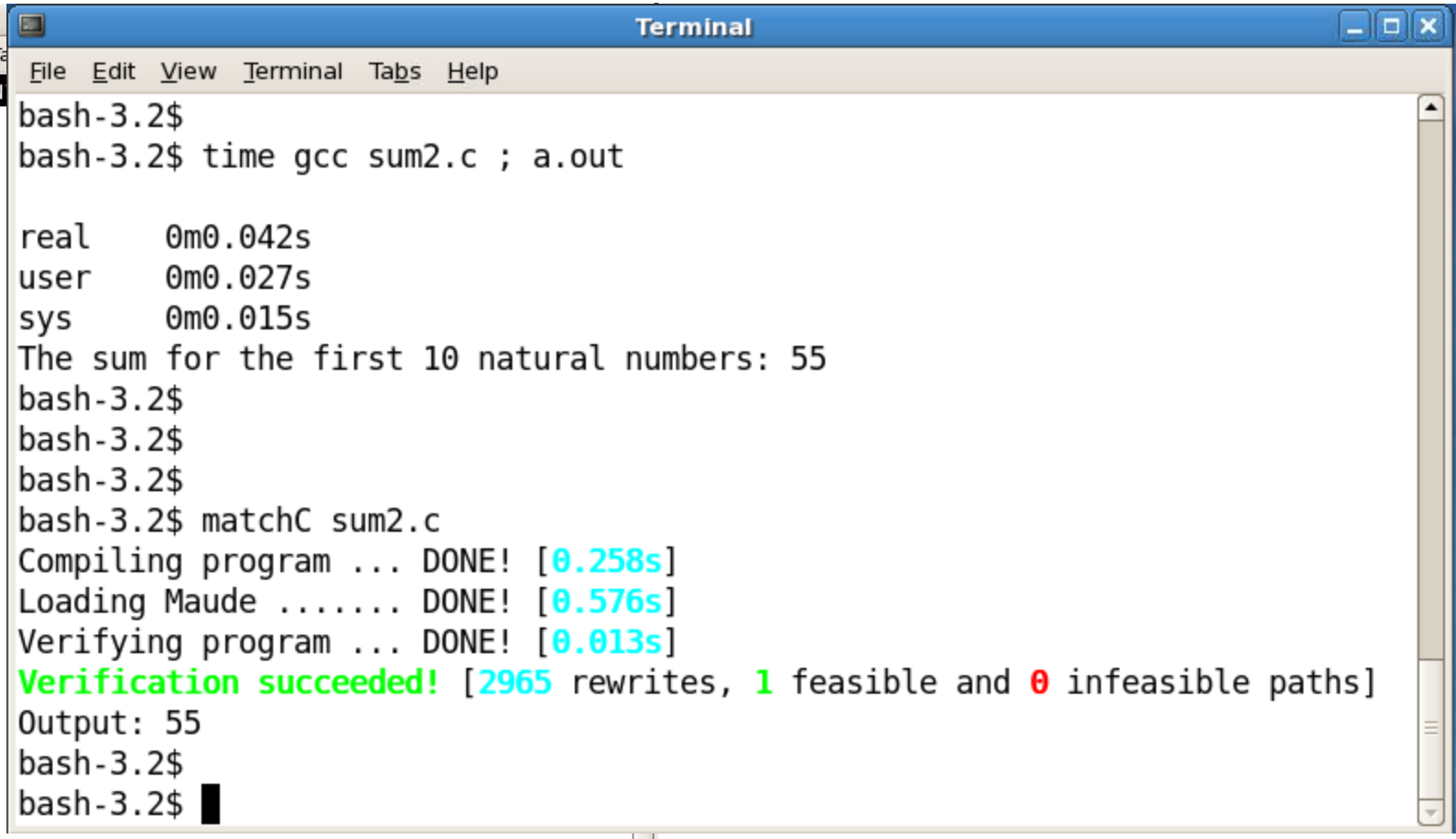
http://fsl.cs.uiuc.edu/index.php/Special:MLOnline

Semantic Execution

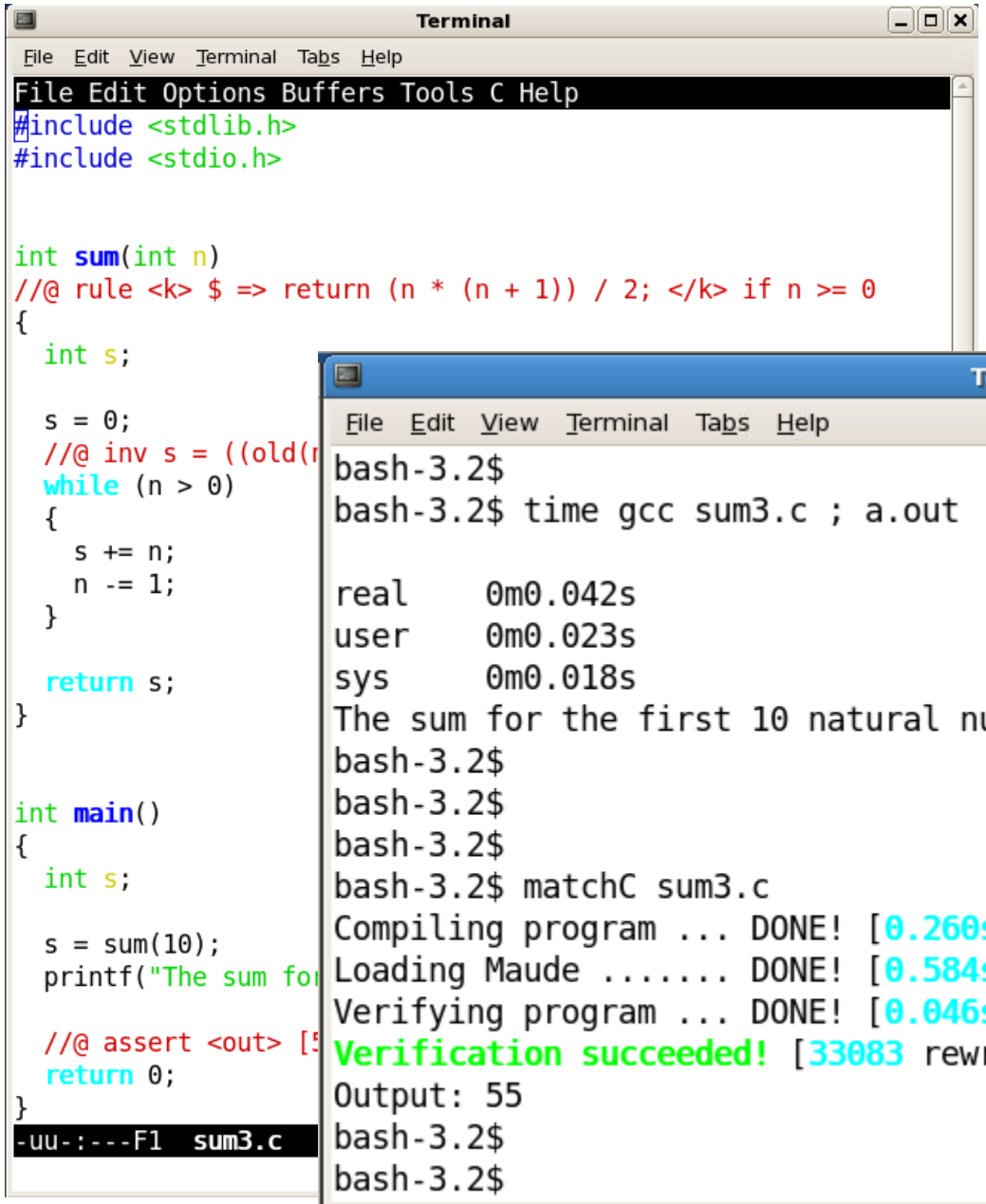


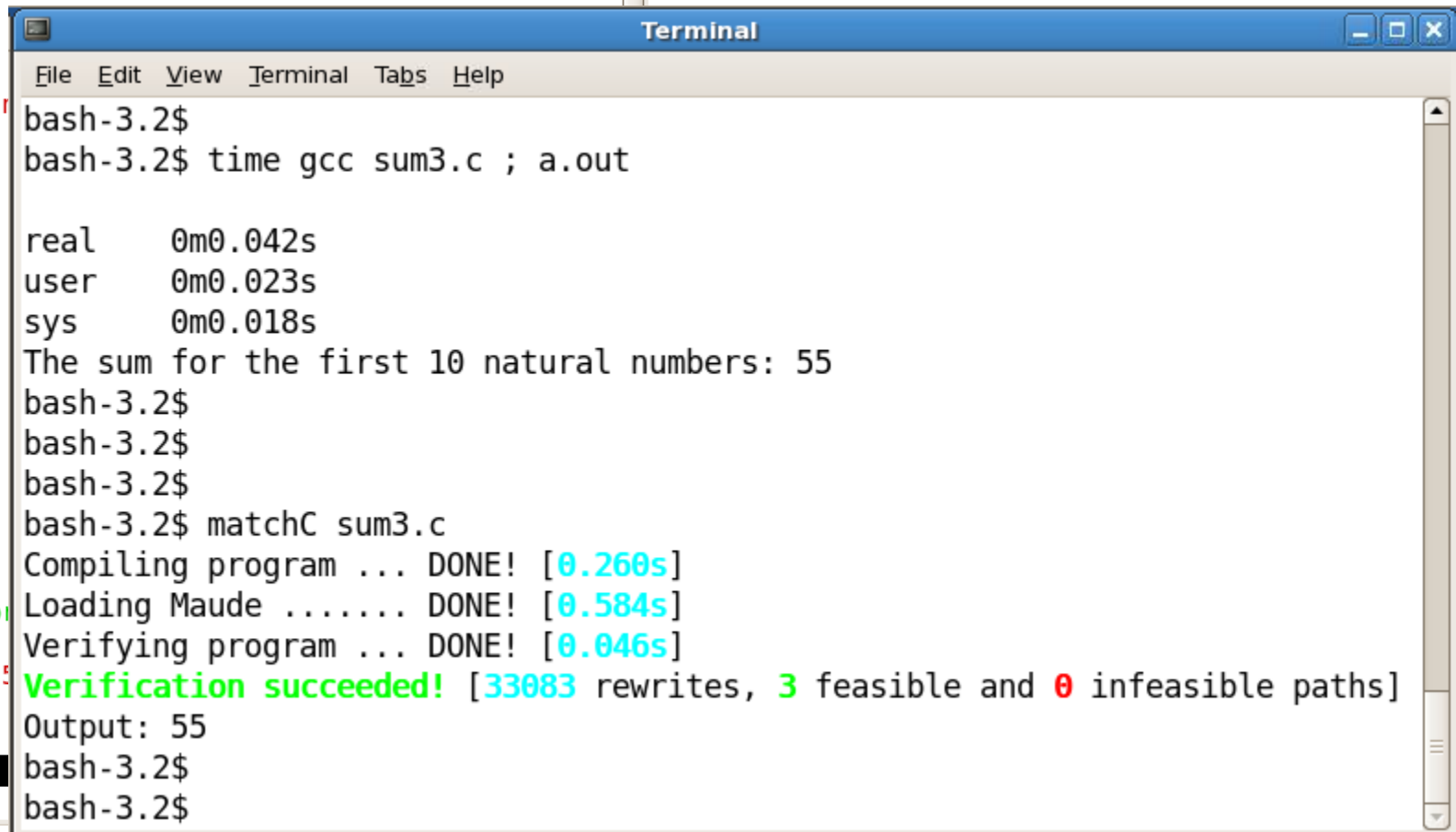
Assertion Checking



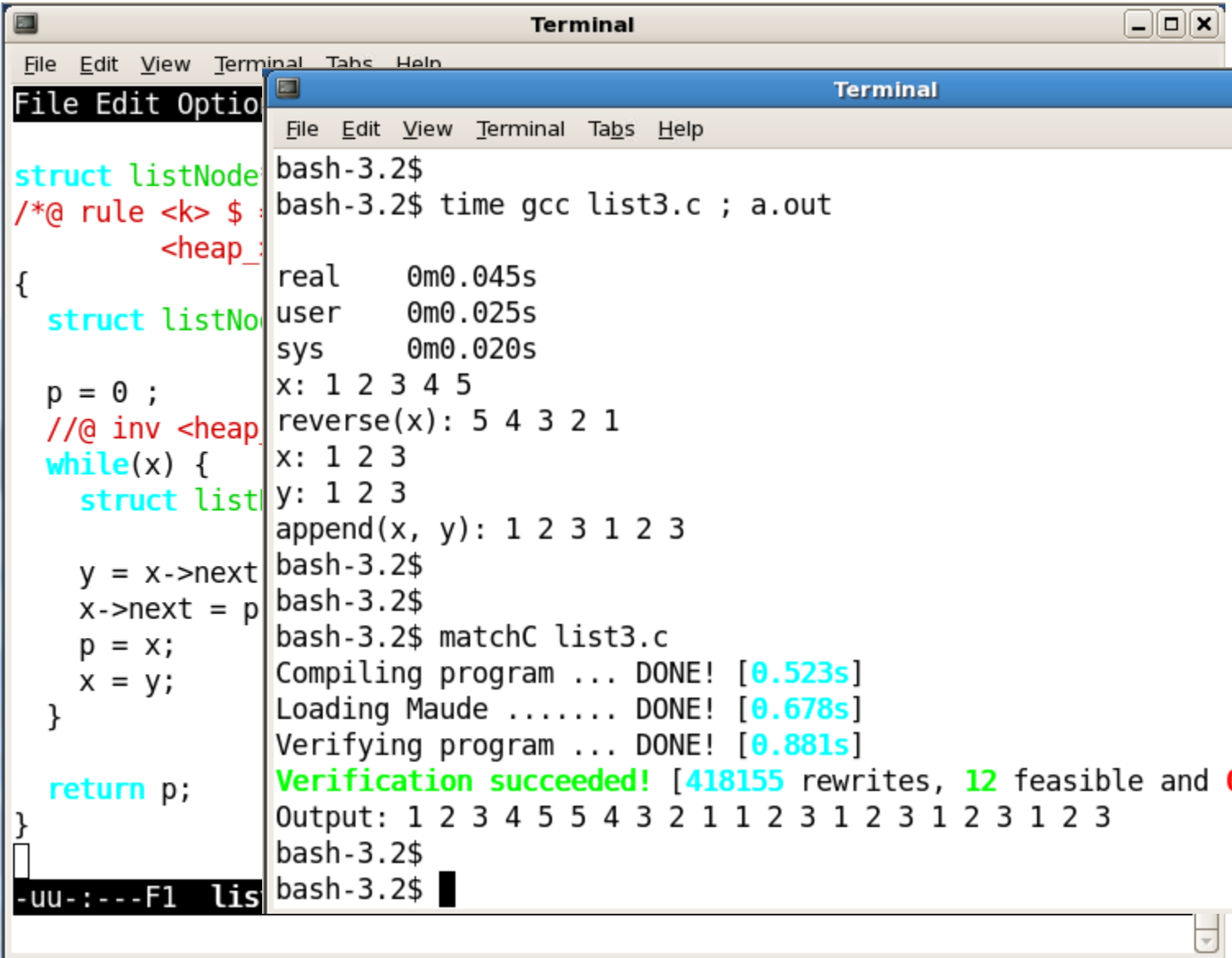


Full Verification

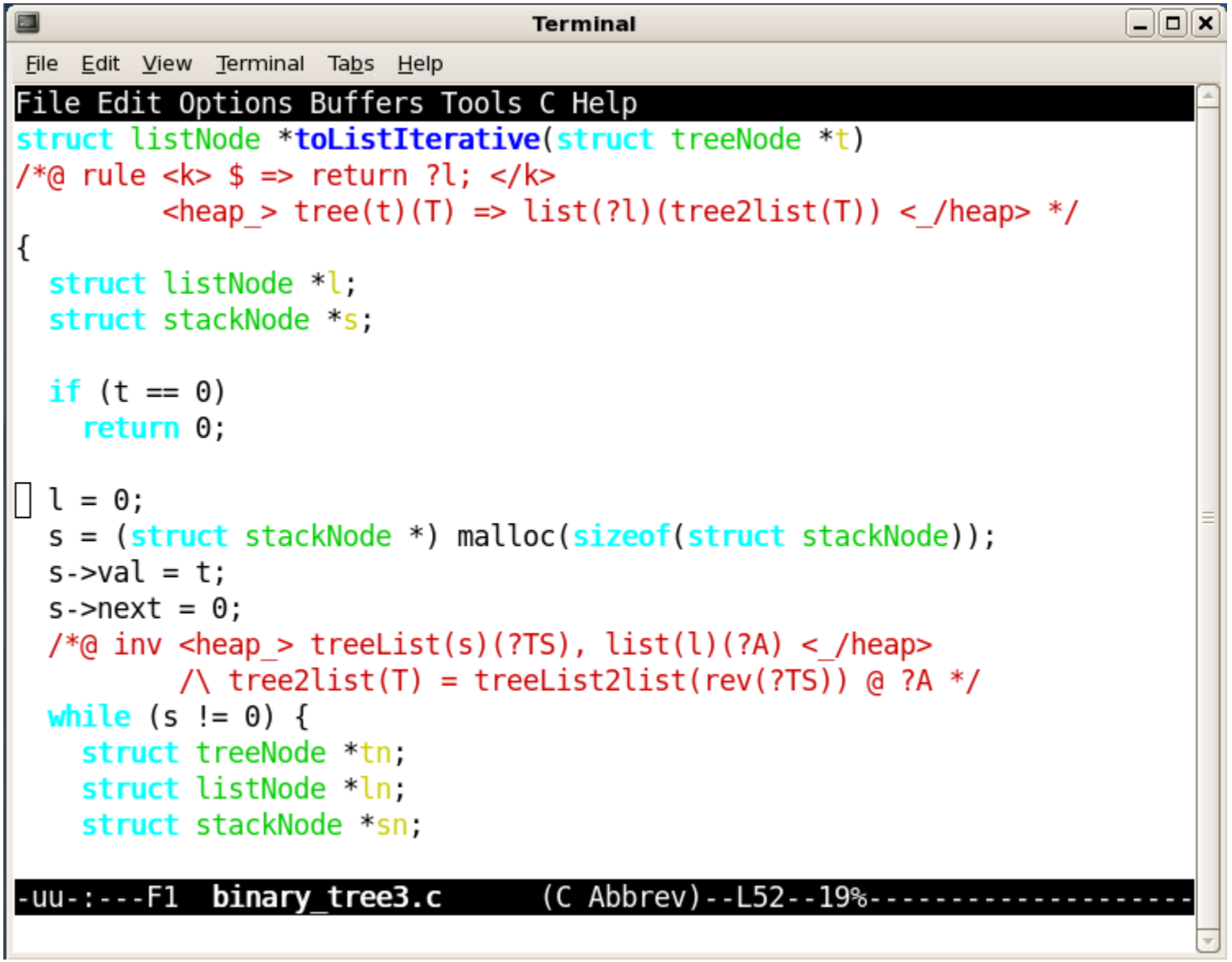




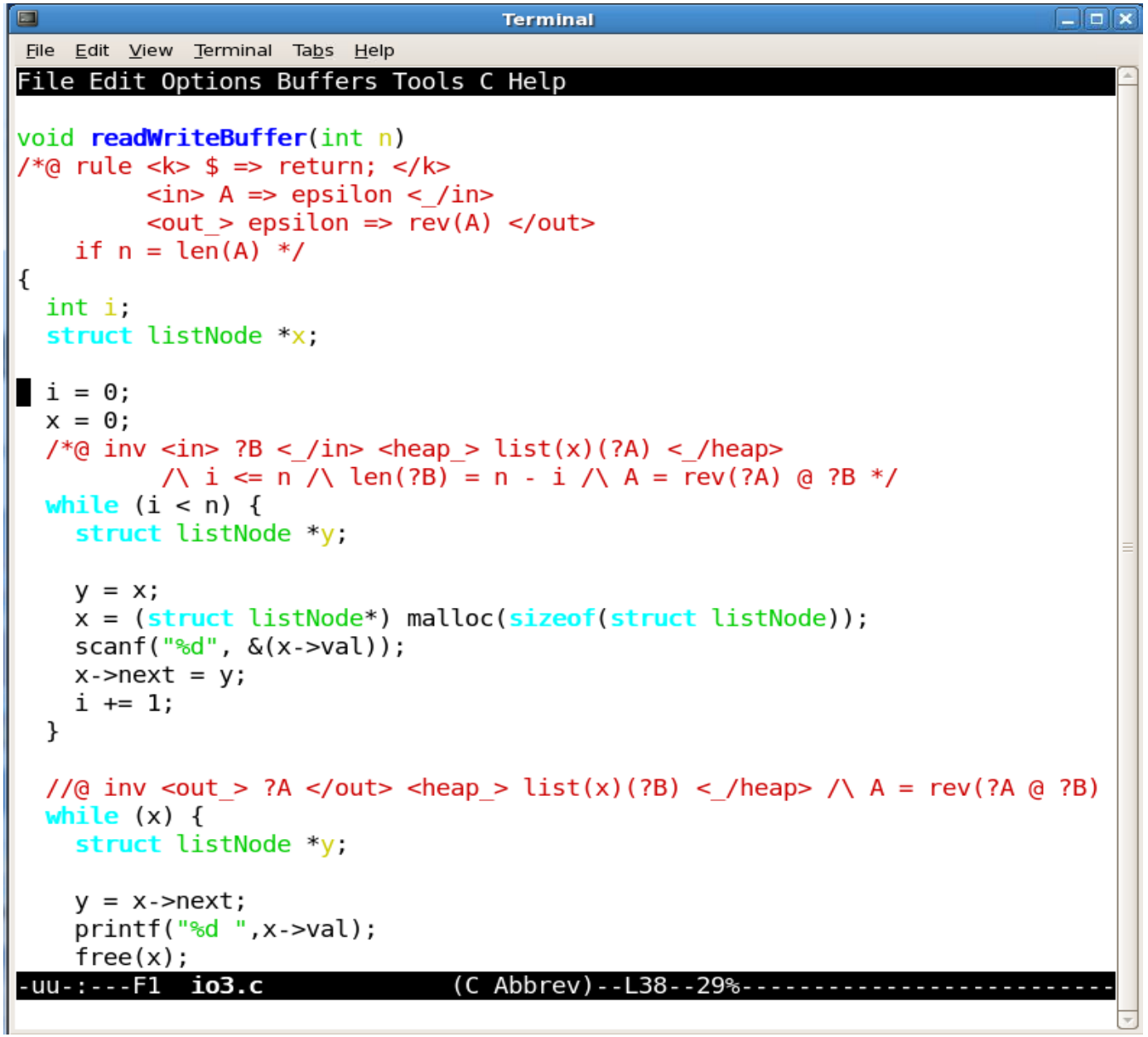
List Examples – Borrowed from SL tools



Beyond Separation Logic Tools



Beyond Separation Logic – I/O



Beyond Separation Logic – Stack Inspection

Terminal

File Edit View Terminal Tabs Help

```
File Edit Options Buffers Tools C Help

void trusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
    if n >= 10 \/ in(hd(ids(S)), {main, trusted}) */
{
  printf("%d ", n);
  untrusted(n);
  any(n);
  if (n)
    trusted(n - 1);
}

void untrusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
    if in(trusted, ids(S)) */
{
  printf("%d ", -n);
  if (n)
    any(n - 1);
}

void any(int n)
{
  // untrusted(n);
  if(n > 10)
    // possible security violated if n < 10
    trusted(n - 1);
}
-uu-:---F1  stack_inspection.c      (C Abbrev)--L14--12%-----------------------
```

Conclusions

- K (semantics) and Matching Logic (verification)
- Formal semantics is useful and practical!
- One can use an executable semantics of a language *as is* also for program verification
  - As opposed to redefining it as a Hoare logic
- Giving a formal semantics is not necessarily painful, it can be fun if one uses the right tools